UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 218

Iterative and Adaptive PDE Solvers for Shared Memory Architectures

HENRIK LÖF

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2006

ISSN 1651-6214
ISBN 91-554-6648-6
urn:nbn:se:uu:diva-7136

Dissertation presented at Uppsala University to be publicly examined in Auditorium Minus, Museum Gustavianum, Uppsala, Saturday, October 7, 2006 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

Löf, H. 2006. Iterative and Adaptive PDE Solvers for Shared Memory Architectures. (Iterativa och adaptiva PDE-lösare för parallelldatorer med en gemensam minnesorganisation). Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 218. viii+49 pp. Uppsala. ISBN 91-554-6648-6.

Scientific computing is used frequently in an increasing number of disciplines to accelerate scientific discovery. Many such computing problems involve the numerical solution of partial differential equations (PDE). In this thesis we explore and develop methodology for high-performance implementations of PDE solvers for shared-memory multiprocessor architectures.

We consider three realistic PDE settings: solution of the Maxwell equations in 3D using an unstructured grid and the method of conjugate gradients, solution of the Poisson equation in 3D using a geometric multigrid method, and solution of an advection equation in 2D using structured adaptive mesh refinement. We apply software optimization techniques to increase both parallel efficiency and the degree of data locality.

In our evaluation we use several different shared-memory architectures ranging from symmetric multiprocessors and distributed shared-memory architectures to chip-multiprocessors. For distributed shared-memory systems we explore methods of data distribution to increase the amount of *geographical locality.* We evaluate automatic and transparent page migration based on runtime sampling, user-initiated page migration using a directive with an affinity-on-next-touch semantic, and algorithmic optimizations for page-placement policies.

Our results show that page migration increases the amount of geographical locality and that the parallel overhead related to page migration can be amortized over the iterations needed to reach convergence. This is especially true for the affinity-on-next-touch methodology whereby page migration can be initiated at an early stage in the algorithms.

We also develop and explore methodology for other forms of data locality and conclude that the effect on performance is significant and that this effect will increase for future shared-memory architectures. Our overall conclusion is that, if the involved locality issues are addressed, the shared-memory programming model provides an efficient and productive environment for solving many important PDE problems.

Keywords: partial differential equations, iterative methods, finite elements, conjugate gradients, adaptive mesh refinement, multigrid, cc-NUMA, distributed shared memory, OpenMP, page migration, TLB shoot-down, bandwidth minimization, reverse Cuthill-McKee, migrate-on-next-touch, affinity, temporal locality, chip multiprocessors, CMP

Henrik Löf, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

ISSN 1651-6214
ISBN 91-554-6648-6
urn:nbn:se:uu:diva-7136 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-7136)

Till mamma och pappa

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I	Löf, H., Nordén, M. and Holmgren, S. Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers. In *Computational Science - ICCS 2004. 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part II*. Lecture Notes in Computer Science 3037, pp. 9–16, Springer-Verlag Berlin Heidelberg 2004.
II	Löf, H. and Holmgren, S. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA system. In *Proceedings of the 19th annual international conference on Supercomputing (ICS05), Cambridge, Massachusetts*, pp. 387–392, ACM Press New York, NY, USA, 2005.
III	Löf, H. and Rantakokko J. Algorithmic Optimizations of a Conjugate Gradient Solver on Shared Memory Systems. In *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, Vol. 21, pp.345–363, Taylor & Francis 2006.
IV	Wallin D., Löf, H., Hagersten, E. and Holmgren, S. Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors. In *Proceedings of the 2006 international conference on Supercomputing (ICS06),Cairns, Australia*, pp. 145–155, ACM Press New York, NY, USA, 2006.
V	Nordén M., Löf, H., Rantakokko, J., and Holmgren, S. (2006) Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. Technical Report 2006-038. Department of Information Technology, Uppsala Universitet, Uppsala, Sweden, 2006. To appear in *The proceedings of the 2:nd International Workshop on OpenMP (IWOMP)*, Reims, France, 2006.

Comments on my Participation

In all of the papers, I contributed to the planning of the research, the design of experiments, the evaluation of the results and the writing of the papers. Details on my participation in the five papers are included below.

I	Markus Nordén and I are the main contributors to this paper. I am one of the co-authors and I designed and performed all of the experiments for the GEMS and the NAS NPB applications.
II	I am the main contributor to this paper. I implemented and performed all experiments.
III	I am the main contributor to this paper. I implemented and performed all experiments.
IV	Dan Wallin is the main contributor to this paper. I am one of the co-authors and I implemented the multigrid solver used for some of the experiments.
V	Markus Nordén is the main contributor to this paper. I am one of the co-authors and I performed all the experiments. I also implemented the load-balancing method used.

Contents

1. Summary in Swedish

Iterativa och adaptiva PDE-lösare för parallelldatorer med gemensam minnesorganisation

TÄNK dig att du är en ingenjör som har till uppdrag att bygga en ny bro. Av specifikationen framgår hur lång den skall vara och vilken slags trafik den skall kunna bära. Hur försäkrar du dig om att din konstruktion kommer att hålla? Ett sätt att undersöka brons hållfasthet är att först bygga den i en dator. Vi bygger naturligtvis inte bron fysiskt utan vi konstruerar en *simulering* av bron och miljön runt omkring den i ett datorprogram. Resultatet blir en virtuell bro som vi sedan kan testa på alla möjliga tänkbara sätt utan att behöva riskera något i den verkliga världen. Samma teknik, d.v.s. datorsimulering, kan användas (och används) för att lösa andra problem som t.ex krocktestning av bilar, flygförmåga och bränsleförbrukning hos flygplan, medicinska egenskaper hos proteiner samt prissättning av optioner.

För att åstakomma en verklighetstrogen simulering behöver vi först en matematisk modell av det som skall simuleras. Sådana modeller utvecklas av t.ex. fysiker, biologer eller ekonomer och för att utföra datorsimuleringen använder man sig i det flesta fall av en s.k. *numerisk metod* som omvandlar modellen till miljontals enklare operationer som lämpar sig väl för datorer. Den numeriska lösningen, d.v.s. datorlösningen till våra modeller lagras i datorns minne och är egentligen bara en approximation till den verkliga lösningen. I den numeriska metoden kan man kontrollera felet man gör i approximationen på olika sätt beroende på vilken nogrannhet som krävs. Man kan tänka sig detta som en "verklighetsratt" som man kan vrida på efter behov. Ju noggrannare simulering vi vill göra destå större och svårare blir simuleringsproblemet. Mängden minne som behövs och antalet beräkningsoperationer ökar ju noggrannare simulering vi vill genomföra. Tyvärr så händer det ofta att de riktigt intressanta problemen blir gigantiskt stora även om vi ställer verklighetsratten i det lägsta läget. För att lösa dessa problem måste vi använda oss av s.k. *parallelldatorer*.

Parallelldatorer med en gemensam minnesorganisation

Den grundläggande idén med parallelldatorer är att vi kan öka beräkningskapaciteten genom att koppla ihop flera enklare datorer och låta dem arbeta tillsammans. Genom att låta flera processorer dela på arbetet kan vi lösa problemet snabbare än med en enda processor. Vi kan också använda det sammanlagda minnet från alla ingående datorer vilket leder till att vi kan lösa större problem. En viktig egenskap hos parallelldatorsystem är att systemet är *skalbart* dvs att ett problem kan lösas *p* gånger snabbare om vi lägger till *p* processorer.

Man brukar dela in parallelldatorer i två klasser: system med ett distribuerat minne[1] och system med ett gemensamt minne[2]. I det distribuerade fallet bildar en eller flera processorer tillsammans med ett lokalt minne en s.k. *nod*. Flera noder sätts sedan samman med ett nätverk till ett större system. I det gemensamma fallet kopplas alla processorerna till samma *gemensamma* minne. En egenskap hos parallelldatorer är att data som används av flera processorer måste kommuniceras mellan processorerna. I det distribuerade fallet måste data flyttas eller kopieras mellan noderna till det lokala minnet för att processorerna skall kunna läsa eller skriva till varandras data. I det gemensamma fallet räcker det med att läsa och skriva direkt till det gemensamma minnet. Sådana system blir enklare att programmera än system med ett distribuerat minne. Problemet är dock att de gemensamma minnet blir en kritisk resurs vilket begränsar skalbarheten i systemet.

Ett sätt att öka prestanda och skalbarhet är att använda sig av s.k. *cacheminnen*[3]. Den senaste tiden har den snabba utvecklingen av processorer lett till ett ökande gap mellan hastigheten på processorn och minnet. Detta *minnesgap* resulterar i att processorn måste vänta på data. För att lösa detta problem lägger man till ett litet, snabbt s.k. *cacheminne* mellan processorn och *primärminnet*. Cacheminnen fungerar p.g.a. att de allra flesta program använder endast en liten del av sitt data intensivt. Denna princip kallas för *datalokalitet*[4] och är en vanlig egenskap hos datorprogram. För maximal prestanda måste alltså våra program utformas så att det utnyttjar cacheminnet på bästa möjliga sätt, dvs att programmet skall uppvisa en hög grad av datalokalitet.

För att ytterligare öka skalbarheten så organiserar man systemet enligt en distribuerad modell, men man lägger till ett system som transparent flyttar eller kopierar data mellan noderna. På detta sätt bibehåller man den attraktiva programmeringsmodellen från system med gemensamt minne fast systemet fysiskt är byggt enligt den distribuerade modellen för ökad skalbarhet. Sådana system kallas för *distributed shared-memory systems*[5] (DSM). En konsekvens att det fysiskt distribuerade minnet är att åtkomsttiden för data blir icke-uniform. Därför benämns DSM-system som *parallelldatorsystem med*

[1]eng. distributed-memory

[2]eng. shared-memory. Ofta kan man också se den svenska översättningen: delat minne.

[3]eng. cache memory

[4]eng. locality of reference

[5]En svensk översättning av begreppet skulle kunna vara: parallelldatorsystem av gemensamt-minnetyp med ett fysiskt distribuerat primärminne

icke-uniform minnesorganisation[6]. Icke-uniformiteten kommer sig av att om vi inte hittar data i något cacheminne måste vi hämta data antingen från det lokala primärminnet i samma nod eller göra en s.k. *fjärråtkomst*[7] för att hämta data från primärminnet i en annan nod via nätverket. Fjärråtkomster blir alltså långsammare än åtkomster till det lokala minnet.

Denna skillnad i åtkomsttid resulterar i att det blir viktigt för prestanda hos programmet att bestämma hur data skall distribueras över noderna. Program som inte genererar stora mängder fjärråtkomster uppvisar en form av datalokalitet som vi kallar för *geografisk lokalitet*[8]. DSM system framstår alltså som en väg att gå för att kunna utföra stora och komplicerade datorsimuleringar. Vi måste dock hantera två problem om vi vill uppnå maximal prestanda: Ett, vi måste maximera datalokaliteten för att minska antalet åtkomster till primärminnet. Två, om vi behöver göra en åtkomst till primärminnet så måste vi minimera antalet fjärråtkomster.

De senaste åren har processortillverkarna insett att den traditionella processorarkitekturen med en enda processor inte kommer att kunna göras snabbare i framtiden. Istället har man valt att placera flera processorer på samma chip s.k. *chip-multiprocessorer* (CMP). Dessa system påminner starkt om de klassiska parallelldatorsystemen men med en viktig skillnad. Eftersom processorerna sitter nära varandra på samma chip kan de kommunicera betydligt snabbare än tidigare parallelldatorer. Flera CMP processorer kan sättas samman och bilda ett skalbart DSM system. För dessa system kvarstår dock problemen med datalokalitet och geografisk lokalitet för att uppnå maximal prestande.

Iterativa och adaptiva PDE-lösare

Vi ser alltså en trend att framtidens datorer kommer att i större utsträckning än tidigare programmeras som en parallelldator med ett gemensamt minne. Denna trend gäller speciellt för svåra simuleringsproblem där programmeringsprocessen är extra komplicerad. I denna avhandling utvecklar vi metodologi för programmering av numeriska lösare av *partiella differentialekvationer* (PDE) för parallelldatorer med en gemensam minnesorganisation. Sådana ekvationer används flitigt i många viktiga simuleringar och har studerats sedan datorns barndom. Vi studerar s.k. iterativa och adaptiva PDE-lösare som används för realistiska simuleringsproblem. För att uppnå maximal prestanda på moderna skalbara system måste vi uppnå en hög grad av lokalitet, både datalokalitet och geografisk lokalitet. Detta åstakommer vi genom optimering och omskrivning av både programkod och algoritmer. Vi söker efter omformuleringar som är generella och utgör små ingrepp i programstrukturen.

[6]eng. non-uniform memory architectures (NUMA).
[7]eng. remote access
[8]eng. geographical locality

I uppsats I undersöker vi först och främst effekten av geografisk lokalitet för fyra olika PDE-lösare. Två av dem är enkla standardprogram och två är mer realistiska simuleringsprogram. De två parallelldatorerna av DSM typ vi använde kunde förbättra den geografiska lokaliteten genom att flytta eller *migrera* data mellan noderna. Hos den ena datorn sköttes migreringen explicit av programmeraren jämfört med den andra där datamigreringen sköttes automatisk genom ett transparent system som övervakade minnesåtkomstmönstren. Vi utforkskar vilken av de två datamigreringsstrategierna som lyckas bäst med att minska antalet fjärråtkomster och vilken inverkan de två strategierna har på programstrukturen.

I uppsats II studerar vi den parallella effektiviteten för den explicita datamigreringsmekanismen för en iterativ PDE-lösare. Vi visar att skalbarheten kan ökas med relativt enkla medel utan att totalprestanda sjunker. Uppsats III behandlar parallell effektivitet och datalokalitet för samma iterativa lösare. Vi utvärderar olika typer av algoritmiska optimeringar och omformuleringar och visar hur prestanda för lösaren kan ökas avsevärt med liten inverkan på programstrukturen.

I uppsats IV studerar vi hur den parallella effektiviteten påverkas av CMP-processorer. Vår ide var att på framtida system av CMP-typ så kommer goda lokalitetsegenskaper vara viktigare än hög parallell effektivitet. Vi studerar en s.k. multigridlösare och visar att en formulering med sämre parallell effektivitet än standardformuleringen löser problemet fortare tack vare bättre lokalitetsegenskaper.

Slutligen så studerar vi i uppsats V en typ av PDE-lösare där den numeriska metoden anpassar sig dynamiskt efter simuleringsförloppet. Sådana, s.k *adaptiva* numerisk metoder är speciellt svåra att parallellisera p.g.a det dynamiska minnesåtkomstmönstret. Datamigrering blir i detta fall nödvändigt eftersom åtkomstmönstret förändras över iterationerna. Vi studerar ett modellproblem och visar att datamigrering förbättrar graden av geografiska lokalitet utan att försämra den parallella effektiviteten hos den adaptiva PDE-lösaren.

Slutsatser

- En hög grad av geografisk lokalitet kan avsevärt förbättra prestanda hos iterativa och adaptive PDE-lösare.
- Datamigrering är en effektiv teknik för att öka graden av geografisk lokalitet för dessa applikationer.
- Programomformuleringar som syftar till att öka graden av datalokalitet kommer att ha en ännu större betydelse för prestanda på framtida system av gemensamtminnetyp än idag.

2. Introduction

Computers are incredibly fast, accurate and stupid; humans are incredibly slow, inaccurate and brilliant; together they are powerful beyond imagination

Albert Einstein

Scientific computing is used to study many important problems in science and engineering ranging from global climate modeling and crash simulations to protein folding and the pricing of stock options. Instead of performing lab experiments or constructing prototypes, scientific computing can be used to *simulate* the real world using a computer program. This methodology is different from classic approaches of conducting science and is being applied in an increasing number of disciplines. A necessary condition for a simulation approach to be efficient is that the computer programs are optimized for the target computer.

Realizing a simulation problem as a computer program requires a mathematical model of the problem studied. The model is transformed into an algorithm suited for computers using *numerical methods*. One fundamental feature of numerical methods is that they compute an *approximation* to the true solution and the quality of the approximation is determined by the detail of the numerical method. As a consequence, very challenging problems requiring detailed models normally result in huge amounts of data that needs to be processed. This means that the usage of a computer simulation methodology is often limited by the size and speed of the computer system used. In many cases it is known how to realize an important simulation, but there is no computer system available that can complete the calculations within a reasonable period of time. In these situations, performance separates what is infeasible from what is feasible. Furthermore, it is not uncommon that numerical simulations only execute at a small percentage (5-20%) of the peak performance of the system. This poor utilization of computer resources is not satisfactory because we want our computing investments to pay off.

Both the feasability and the utilization problem can be attacked using various software and algorithmic optimizations. Optimizing the code can enable very large simulations previously out of grasp or make it possible to increase the throughput and efficiency of simulation codes already in use. If the factors that determine performance are known it may also be possible to trade performance for other important characteristics of software such as modularity, readability, extensibility, ease of use and robustness.

In this thesis we explore and develop methodology for performance optimization of simulation code for a class of computer systems referred to as *shared-memory architectures*. These architectures have the potential of combining high performance with ease of programming which simplifies the solution of more challenging simulation problems. We study numerical methods for *partial differential equations* a.k.a. PDE which are used frequently in simulations and we evaluate ways of implementing PDE solvers for a range of shared-memory architectures. The PDE problems studied are iterative or adaptive in nature and they exhibit data locality problems for shared memory architectures. This is especially true for non-uniform architectures and we address these locality problems using algorithmic optimizations. We start by providing some background to shared-memory architectures, programming and the implementation of PDE solvers. Our aim is to provide the terminology and concepts needed for the presentation of our contributions presented in Chapter 3, and Chapter 4. We conclude the thesis summary with an outlook on the future in Chapter 5.

2.1 Shared Memory Architectures

A computer lets you make more mistakes faster than any invention in human history - with the possible exceptions of handguns and tequila

Unknown

Shared-memory architectures have been used extensively in commercial and scientific domains. The key property of these systems is that the processors all share the available resources such as memory and I/O capabilities. This design allows for very flexible resource sharing and adaptation to many different types of workloads. For scientific computing applications shared-memory architectures need to be able to efficiently support a large processor count without becoming too expensive. To achieve this, manufacturers of large-scale shared-memory systems have relied on two major ideas: adding cache memories to hide memory latencies and distributing memory to cope with high bandwidth demands.

2.1.1 Cache Memories

The speed of memory has not developed in pace with the memory bandwidth demands of modern microprocessors. Large, fast memories are simply too expensive or complex to build. It is more cost efficient to create the illusion of a fast memory by inserting a small but fast *cache memory* [45, 50] between the large, slow main memory and the processor (CPU). A request for a word can either hit or miss in the cache. On a miss the request is forwarded down to the slower main memory. For cache memories to be efficient, programs must be able to keep their critical data in the cache memory to a high degree. We call this principle *locality of reference*.

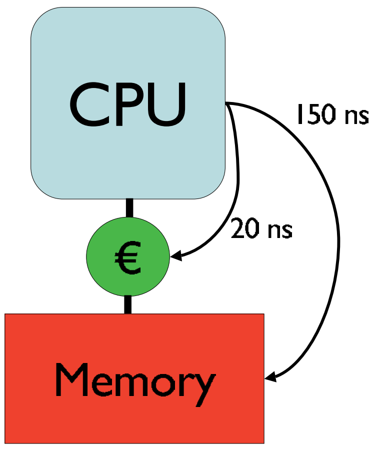


Figure 2.1: Example for illustrating cache effects. The access time to a cache memory (the "euro" sign) at a cache hit is 20 ns and the access time to main memory at a cache miss is 150 ns.

As an example, consider the situation sketched in Figure 2.1. In this example, the access time to the cache memory is 20 ns and the access time to the main memory is 150 ns. Assume that 10% of our memory references go to the cache memory, i.e. that we have a *cache hit ratio* of 10%. In this case our average access time is $0.1 \times 20 + (1 - 0.1) \times 150 = 137$ ns. If we somehow increase our cache hit ratio to 90%, our average access time reduces to $0.9 \times 20 + (1 - 0.9) \times 150 = 33$ ns. The point of this example is to show that, if the application exhibits a high cache hit ratio, we can build a less-expensive system with a much lower average access time to memory using caches.

Every cache memory has a finite capacity, and we need an efficient mechanism to determine if data is in the cache or not. This is described by the so-called *cache organization*. In efficient cache organization, data is brought into the cache in larger blocks (or lines), which means that every time we reference a data word we store a whole block of data in the cache memory. As a consequence, access patterns featuring unit strides are efficient on systems with caches. Instead of missing on the references to every word of the block, we generate only one miss per block. All subsequent accesses to this block can be satisfied from the cache memory.

One way to reduce the amount of cache misses is to use a larger cache memory. Today, most of the chip area of microprocessors is used for cache memories. However, large caches also increase the overhead for finding and replacing data in the cache. As a result, modern computer designs use a hierarchy of cache memories of increasing size, referred to as the *memory hierarchy*. Building an efficient memory hierarchy requires several difficult design trade-offs with respect to efficiency, design complexity, chip area, and power requirements. Once this optimum is found, the responsibility of localizing the memory accesses is delegated to software. By optimizing our programs we can exploit a given cache organization better.

2.1.2 Multiprocessors

A traditional way to increase performance further is to connect several processors together using a network to form a *multiprocessor*. The processors can then cooperate to finish a task faster than a single processor. Using a naive model, if one processor can finish the task in T_1 time, p processors can finish the task in

$$T_p = \frac{T_1}{p} \tag{2.1}$$

time. This model is an example of so-called perfect *scalability*. Another way of looking at scalability is to state that by increasing p we can solve a problem p times as large as the original problem in the same amount of time. As we will see in Section 2.2 such perfect scalability is very hard to achieve.

There are two main classes of multiprocessors: *distributed-memory* and *shared-memory* architectures [28, 35, 46, 50]. In shared-memory architectures, all processors share a single global memory. Writes to this shared memory are visible to reads for all processors of the system. In distributed-memory systems, each processor has its own local memory, which cannot be accessed directly from another processor. Writes to this local memory are not visible to reads from other processors. This distinction has profound implications on how the two types of systems are programmed.

The advantage of distributing the memory is that we can improve memory bandwidth since the memory requests are distributed over several memory controllers. However, most parallel programs require communication. On distributed-memory architectures, communication is often realized using a *message-passing* programming model [28, 35, 46]. Messages are sent between the processors to synchronize the local memories, thus allowing writes to be visible. By explicitly controlling the messages we gain control over the interconnect network traffic. However, this control comes with the drawback of complex programming, which might reduce overall productivity.

For shared-memory multiprocessors, communication between the different processors is handled in a more transparent way. A programmer can simply read and write to the shared memory. However, because we need cache memories for bridging the CPU-memory gap we are faced with another problem. Cache memories may hold local copies of data, and if some data is shared between processors, this leads to a *cache-coherency* problem [28, 50]. To solve this problem a cache-coherency protocol is implemented to maintain coherency of shared data by invalidating or updating incoherent copies. At the next request to an incoherent copy, the most recent data needs to be communicated to the requesting cache memory. The use of cache coherency protocols leads to a situation where the amount of interconnect traffic is again (like locality of reference) governed by the access patterns of the application, and by controlling the access patterns we can improve performance of applications.

2.1.3 Distributed Shared Memory Systems - DSMs

There exist shared-memory multiprocessors that have physically distributed memory, but they can still be programmed as a shared-memory system. This is achieved by adding a transparent communication layer that provides the illusion of a single shared memory. Shared-memory architectures are usually classified into two different categories based on this distinction.

Symmetric Multiprocessor (SMP) In these architectures, the access time to the shared memory is uniform. Hence, they are classified as *uniform memory architectures* (UMA). These architectures represent typical, small-scale shared-memory architectures on the market today and they usually exhibit processor counts of a few dozen.

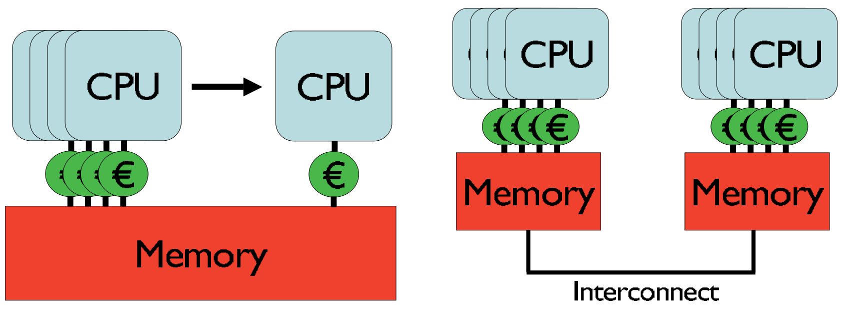
Distributed Shared Memory Architectures (DSM) In these architectures, the memory is physically distributed over a set of *nodes*. As a consequence, the access time to the logically shared memory is non-uniform. Hence, these architectures are classified as *non-uniform memory architectures* (NUMA). DSMs are constructed to support the bandwidth demands of a larger amount of processors without incurring excessively long access times. Cache-coherent DSMs are often referred to as cc-NUMA systems.

The idea of DSMs is to combine the high bandwidths of distributed architecture with the shared-memory programming model. Ideally, such systems represent a cost-efficient way of building large-scale shared-memory systems. By using standard building blocks for the individual nodes, a large-scale system can be built using some high performance interconnect at a lower cost compared to a centralized SMP design [28, 50, 97].

In a DSM architecture, accesses that miss in the cache need to be retrieved from the physical memory of some node. If the cache miss can be served by memory from the same node, we call it a *local access*. Otherwise, the data must be fetched from the memory of another node, and we call this type of access a *remote access*. Because remote accesses need to travel through the interconnect network, the access time for memory within a node is smaller than the time required for a remote access. The difference is measured by the *NUMA ratio*.

$$\text{NUMA ratio} = \frac{\text{Remote access time}}{\text{Local access time}}. \tag{2.2}$$

Typical NUMA ratios range from 2 to 10 depending on the system and network architecture. Apart from the previously discussed form of locality of reference, DSM architectures add an additional form of locality. The *geographical locality* of data describes the amount of remote accesses issued to the memory system, and an application in which most accesses are served by local memory is said to exhibit a high degree of geographical locality.



(a) Symmetric Multiprocessor (b) Distributed Shared-Memory System

Figure 2.2: Illustrations of the two major types of shared-memory architectures. Figure 2.2(a) depicts a typical shared-memory architecture where every CPU have the same uniform access time to the shared memory (UMA). Figure 2.2(b) depicts a distributed shared-memory architecture (DSM) in which the physical memories are distributed over a set of nodes. The CPUs will see different access times depending on the node affinity of the data. Hence, such architectures are classified as non-uniform (NUMA).

Many applications exhibit large working sets, which lead to cache capacity problems. To reduce the latency for the cache misses, several architectural improvements have been proposed. Data distribution via page placement is discussed in Section 2.2. Other techniques include *Remote Access Caches (RAC)* [65, 106] and *Cache-Only Memory Architectures (COMA)* [48], *Simple COMA (S-COMA)* [47, 49, 103], and *Reactive NUMA (R-NUMA)* [41, 65].

2.1.4 Chip Multiprocessors - CMPs

Recent developments in microprocessor architecture suggest that future shared-memory systems will be hosted on a single chip [2, 91]. Chip Multiprocessors (CMP) [7, 50, 61, 62, 63, 64, 79, 112] are constructed by placing several microprocessors, so-called processor *cores*, on a single chip and connecting them using an on-chip network. Each core can also be capable of executing code from several programs simultaneously, a technique called Simultaneous Multi-Threading (SMT) [39, 50, 58, 61, 62, 76].

Combining these two techniques will produce shared-memory architectures with drastically different memory system characteristics because the cores can communicate on-chip. Because processor cores are replicated on chip, new design tradeoffs needs to be made. The percentage of chip area used for cache memories will probably decrease, which will emphasize the importance of locality even further. Whether these systems will be uniform or non-uniform is not clear today. However, several CMP chips can be connected together using interconnect hardware to form a large shared-memory architecture. Because each one of these chips host many CPUs, the chips will need very high bandwidth interconnects. Hence, it is likely that they will be built like a DSM [7, 77].

CMP architectures will increase the general availability of shared-memory systems. In just a few years, every new desktop computer will be able to execute two threads or more in parallel. Hence, we believe that there will be an increased demand for parallel programs [109, 110]. In the light of these observations, the shared-memory programming model seems like an attractive abstraction for producing future parallel programs.

2.2 Parallelization and Shared Memory Programming

There ain't no such thing as a free lunch.

R. A. Heinlein

Parallelizing a sequential program and producing an efficient implementation is a non-trivial task. First, we must identify a set of independent tasks that can be executed in parallel to solve a given problem. These tasks then need to be assigned to threads, which are mapped to the processors. Our ultimate goal is to reach the holy grail of parallel computing, or perfect parallelism, which as stated in Equation 2.1. Unfortunately, we can only reach this level of parallelism for a very small amount of applications. In all other cases, the amount of parallelism will be limited by various forms of *parallel overhead* [3].

It can be very hard to track down the reasons for parallel inefficiencies because so many complex subsystems of the computer are interacting. This is especially true in the shared-memory model in which many of the mechanisms generating overhead are hidden from the explicit control of the programmer. In the following sections we will give a brief overview of how to program in the shared-memory model. We will also try to explain what the potential sources of parallel overhead are, especially for non-uniform architectures. Finally, we will discuss cross-cutting issues related to the operative system.

2.2.1 An Introduction to Shared Memory Programming

The fundamental abstraction in the shared-memory programming model is the concept of a *thread*. Multiple threads share the address space of a program and enable concurrent and parallel execution. Although the address space is kept coherent, we often need to synchronize the threads to ensure the correctness of the program. This is a characteristic feature of the shared-memory programming model [35, 46]. Unprotected accesses to the same shared data item will cause a so-called *race condition* or *data race*.

The POSIX [25] standard provides a standardized interface for programming in the shared-memory model. In many cases, the POSIX standard often becomes too detailed and cumbersome to use. To make it easier for the programmer to handle the synchronization and creation of threads, a consortium of vendors designed the OpenMP language extensions [31, 78, 92].

In OpenMP, a master thread executes until it encounters a parallel region. At this point, several threads are created, and the work contained in the parallel region is divided onto the threads using so-called *work-sharing directives*. This style of programming is often referred to as *fork/join* programming. In OpenMP, the parallel regions are identified by inserting a directive or compiler pragma in the code. This method has the advantage that the code can be incrementally parallelized by adding more and more directives. Also, the directives can be ignored using compiler flag, resulting in a serial program. OpenMP was designed to increase portability and productivity compared to previous directive-based languages and models [8, 93]. To achieve these goals the language needs to be both easy to use and architecture-agnostic. Today, OpenMP is the de facto standard for shared-memory programming. Examples of other contemporary shared-memory languages include *PARMACS* (*ANL*) macros [4], *Universal Parallel C* (UPC) [18, 40], *Co-Array Fortan* [89], *Titanium* [126], and *Cilk* [15].

2.2.2 Sources of Parallel Overhead

To produce an efficient parallel implementation in the shared-memory model we must control the three most important sources of parallel overhead: communication, artifactual communication, and synchronization. As a programmer we can accomplish this by targeting various forms of optimizations such as maximizing data locality, minimizing the volume of data exchange, and minimizing the frequency of interactions.

Communication

The classic source of parallel overhead is communication. In the shared-memory model, communication is not explicitly handled by the programmer. Instead, threads communicate by reading and writing to the shared address space, illustrated in Figure 2.3. For communication to take place, thread 1, T_1, must make sure that the write event occurs before the read event of T_2; in other words, a *causal relation* must exist between the two memory events. This relation is enforced using *synchronization*. Figure 2.3 is an example of a communication pattern referred to as *true sharing*, because the value of variable a is used by both threads.

Like any other parallel computer system, the speed of communication is limited by the characteristics of the interconnecting network, and if these limits are exceeded, the network will be congested. Another problem is that interconnect networks can be saturated if the communication occurs in bursts. Of course, we cannot simply remove communication, because it is necessary for producing correct results. However, we can overlap communication with

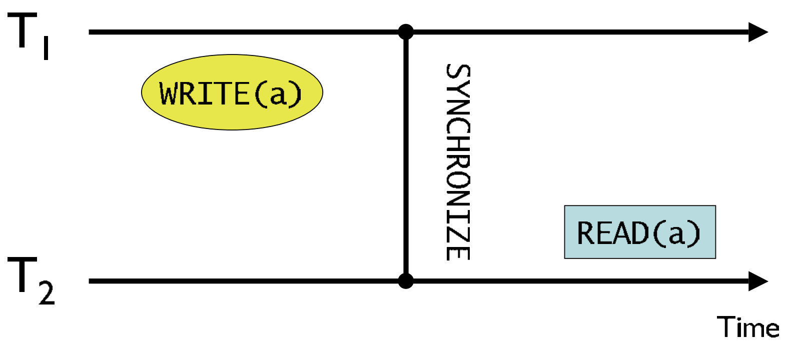


Figure 2.3: Illustration of communication in a shared-memory architecture. The threads T_1 and T_2 communicate using the shared variable a and a synchronization operation.

computations to hide the associated overhead by exploiting the *memory consistency model*, see Culler et al. [28].

Artifactual Communication

A problem related to communication is that many shared-memory systems introduce *artifactual* communication due to the cache organization and coherency protocol. The shared address space is normally kept coherent at some non-word size unit, typically at cache block granularity. If data is shared between two threads, this can lead to a communication pattern referred to as *false sharing*. As an example of false sharing, consider the case in which one thread writes to the first word of a cache block. If a second thread writes to another word in the same block, coherency must be enforced even though the threads are not communicating. This will introduce unnecessary overhead from the coherency protocol. To solve the false-sharing problem, special care has to be taken when designing data structures.

Synchronization

A final source of parallel overhead is attributed to the synchronization operations used to enforce causal relations. As an example, a *barrier* is used to stall all threads until they have reached the same point in the program. Other types of synchronization operations are *reductions*, used to broadcast data, and *mutual exclusion* operations, needed to avoid data races.

Synchronization operations will generate communication overhead, because they are normally implemented using memory operations. However, the primary reason for synchronization overhead is associated with *load balance*. Consider this barrier operation: if the computational work is unevenly balanced, the most loaded thread will force all other threads to wait at the barrier. By assigning an equal amount of computational work, the barrier stall time will be reduced. Note that a perfect computational work distribution may not result in a perfect load balance because of cache effects.

Moreover, on non-uniform architectures, such cache effects are exaggerated because the memory access time is also affected by the physical location of data.

2.2.3 System Support for Increasing Geographical Locality

To achieve a high degree of geographical locality on DSM architectures, we can either optimize our program code for a given architecture or we can design systems that transparently optimize thread-data affinity. There have been many proposals of features to improve the geographical locality of applications, and we present and discuss some of them below.

Thread Scheduling

When a new thread is created it gets scheduled onto one of the DSM nodes. On a time-sharing OS, threads might also be moved (migrated) to some other node to balance the workload. Moreover, on a DSM system it is normally beneficial to include the notion of affinity in the scheduling algorithm. Affinity scheduling means that the thread should be scheduled to the same processor or node on which it had been previously executing. Affinity scheduling increases the possibility of data reuse from caches and local memory [20, 27, 85, 117].

Page Placement Policies

As with thread scheduling, the OS also needs to decide how to distribute the application working set over the nodes. This introduces the notion of *page placement policy*. A memory *page* is the smallest unit of memory allocation used by the OS. The most common page placement policy on contemporary DSM systems is the *first-touch* policy [75], where pages are placed on the node where a thread first references the page, or where it generates a page fault. Other policies include *random placement*, whereby pages are distributed randomly in a uniform fashion, and *round-robin*, whereby pages are placed in a modulo-*n* fashion. *Skew-mapping* and *prime-mapping* schemes are modifications of the round-robin policy [53]. The first-touch policy has been shown to work satisfactory for some numerical applications [75, 81, 82, 83, 84, 117]. First-touch also works well for single-threaded programs.

If the first-touch strategy is used, affinity scheduling becomes important. If threads are moved from one node to another, the amount of remote accesses will increase, because data is physically allocated to memory in the node where the thread first touches it. The optimal schedule would be to keep the thread in the node where it was first scheduled. Some systems also include ways of controlling the placement of pages [12, 43, 66], and in these systems a programmer can explicitly distribute pages to better match the access pattern of the application.

Page Migration

In many cases, an application exhibits phases in which the memory access pattern changes significantly. For example, it is common that a single thread initializes the data before more threads are created. On a system with first-touch page placement policy, serial initialization will result in all data being allocated to the node where the thread executes. In the parallel phase, threads not scheduled to this node will generate a large amount of remote accesses. This problem can sometimes be solved by initializing the data in parallel and ensuring that threads are not moved. However, for more complex codes, it can be very hard or even impossible to initialize data in parallel.

A way of adapting to changes in the access pattern is to try to automatically migrate pages to minimize the amount of remote accesses, or, in other words to move the data to the threads. Such mechanisms can be constructed in user-level mode, by explicit page placement calls [12, 20, 74, 82, 113] or by augmenting the operative system with dynamic data-distribution features [47, 66, 106, 117]. Page migration has shown to be efficient in many scientific applications [16, 27, 51, 65, 82, 87, 106, 117] and in commercial software [47, 117, 123]. On the other hand, some researchers have reported that page migration does not improve performance: see Chauhan et al. [22] and Jiang and Singh [55, 56]. In this context it should be noted that if the working set of the application is too small, most of the remote accesses will be satisfied by local caches. The same application can exhibit very different characteristics depending on the problem size [52, 55, 56]. Many previous studies have also been performed using simulated systems [19, 52, 65, 106, 117, 123], and to make such studies feasible, the working sets need to be small. Otherwise, simulation will take too long. Hence, the architectural parameters need to be carefully scaled down to match the reduced working sets. Another problem of conducting page migration studies is that some applications of common benchmark suites have already been optimized for a specific page placement policy such as first-touch. As discussed earlier, the OS scheduler also needs to be affinity aware. There are examples of the native OS scheduler not being well tuned to the migration mechanism: see Corbalan et al. [26, 27], and Nikolopoulos et al. [85].

Designing heuristics and algorithms for the migration engine is not an easy task and it is dependent upon the architectural support. A migration engine needs to detect candidate pages for migration. For this purpose, various metrics have been studied, such as page reference counters [66, 113], sampling various forms of cache misses [47, 117], DTLB misses [74, 117], and/or combinations of the previous three. The sampling can be completely transparent using hardware mechanisms or be introduced by some form of run-time profiling from instrumented code [74, 81].

Most migration schemes are based on some form of access statistics (counters), which means that a page is migrated if the number of remote accesses rises above a given threshold. If this threshold is set too high, migration might not be triggered at all, or be triggered too late. If the threshold is set too low, pages could ping-pong between nodes incurring substantial overhead.

Once we have an efficient detection mechanism, we also need to control the overhead associated with moving pages between nodes. The two major sources of overhead can be attributed to keeping TLBs coherent [111] and performing the actual copying. Verghese et al. [117] found that TLB flushing and processor synchronization were the primary sources of overheads. The SGI Origin systems include support for reducing this overhead [66]. Typically, the algorithms for TLB consistency does not scale because a TLB shoot-down operation is a global operation [65, 106].

Page migration can also be initiated from the application. Such an explicit mechanism has the advantage of allowing the programmer to control *when* to migrate and adapt to changes in the application. By enabling migration at an early stage, the overhead can be fully amortized. This is especially important in iterative codes [80, 82, 84, 86]. A recent method of implementing such an explicit mechanism is to include a call or directive with an *affinity-on-next-touch* semantic [12, 80, 87]. Using this system call, a programmer can redo a placement using the first-touch policy again. This mechanism can also be used to build a transparent migration engine: see Tikir and Hollingsworth [113].

Many of the features described above have been implemented in existing systems. The SGI Origin systems [66] were one of the first systems of this type on the market. It supports a first-touch placement policy as well as a dynamic page-migration mechanism. These systems also feature system calls for performing explicit page placement. The HP AlphaServer GS Series [12, 43] supported mechanisms for explicit page placement as well as an affinity-on-next-touch call. The Sun WildFire [47, 50] prototype supports dynamic page migration and S-COMA page replication engine, which has been proven to work very well for many applications, [16, 47, 51, 87]. The Sun Fire series of servers [21], used in our experiments, supports a choice of random or first-touch placement policy and an affinity-on-next-touch call [108].

2.3 Implementing PDE Solvers for Shared Memory Architectures

The challenge is not to make easy programs easier.
The challenge is to make hard programs possible

William Gropp

Partial differential equations are used to model many important phenomena. Most interesting PDEs cannot be solved using analytical methods. Instead, numerical methods such as the finite-difference method (FDM) or the finite-element method (FEM) must be used to *approximate* the solution of the PDE using a finite set of points or a finite set of basis functions, a process referred to as *discretization*. A *mesh* or *grid* is used to represent the discrete entities from discretization, and the resolution of the grid determines both the accuracy of the approximation and the amount of memory needed to store the solution. Accuracy can also be improved by the approximation properties of the numerical method itself using so-called *high-order methods*. The choice of numerical method depends on many things: the type of PDE, the boundary conditions, and the geometry of the problem. From an implementation point of view, the choice of mesh has the most profound implication on the potential performance of the simulation code.

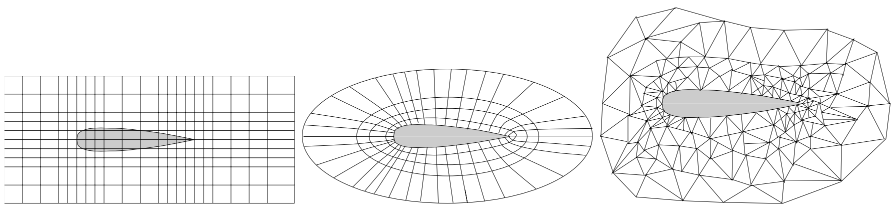

(a) Cartesian Grid (b) Curvilinear Grid (c) Unstructured Grid

Figure 2.4: Examples of different grid types used in the discretization of PDE problems. In the figures, the space surrounding an airplane wing-profile is discretized.

There are two basic types of grids. *Cartesian grids*, often connected to FDM, can be described as a set of starting points and step lengths. As an example, a discretization of the closed interval $[a,b]$ can be generated at a resolution of N points by the function $x_i = a + ih$ where $i = 0 \dots N-1$, and $h = \frac{b-a}{N-1}$. In this formulation cartesian grids can only describe rectangular geometries, see Figure 2.4(a)[1]. However, by applying a coordinate transformation, cartesian grids can be generalized to *curvilinear grids* see Figure 2.4(b). Such grids can be used to cover more general shaped but still regular domains. The combination of several overlapping grids to form a composite grid using interpolation can also be used to handle more complex geometries [95]. *Unstructured grids*, used primarily for FEM, can be described as a subdivision of a given geometry using simple geometric shapes such as triangles or quadrilaterals, see Figure 2.4(c). Such grids can be made to conform to almost any geometry which is a major advantage compared to the cartesian approach. Cartesian grids on the other hand, are easily mapped to a computer memory using array data types. For unstructured grids, all points and edges need to be stored into a data structure for lookup as the grids are traversed. The mapping to a linear memory structure is more complex compared to the cartesian case.

[1]Figures 2.4(a),2.4(b) and 2.4(c) are reprinted from Pärt-Enander [95] with kind permission from the author.

2.3.1 Iterative and Adaptive PDE solvers

Most numerical methods are iterative in nature. For time-dependent PDE problems, the iterations correspond to stepping through time in discrete quanta. Other PDE problems are iterated until some steady-state is reached. Many numerical methods also include additional iterative schemes for each time step or steady-state iteration. This is especially true for more challenging simulation problems that are non-linear or that exhibit phenomena at different time or space scales. This layered approach creates a setting where there are several nested iterative algorithms present. The iterations can be static in the sense that the access patterns does not change over the iterations. In this case the same grid or set of grids is used during the whole simulation. In *adaptive* methods, the access patterns are more dynamic. Such methods modify the grid frequently to resolve phenomena in the simulation as they evolve or as they are identified.

When optimizing iterative algorithms, we need to reduce the total amount of iterations to accelerate convergence, and also minimize the time needed to perform an individual iteration. In this thesis we are concerned primarily with the second type of optimization, and such optimizations have been studied extensively for many computer systems. However, for non-uniform shared-memory systems, most of the previous work has been performed using simpler benchmarking codes. We are interested in optimizing more realistic problems with more complex program structures and/or large datasets exhibiting more dynamic, and irregular access patterns. In realistic settings many practical considerations must be taken into account when optimizing code, and optimizations developed for simpler codes may not be applicable, or they can be hard to implement. Our results are based on three PDE codes which we believe represent realistic solvers used for large scale simulations. We begin by discussing the GEMS application which is studied in Paper I-III. We then continue to the TBGS application (Paper IV), and end with the SAMR application (Paper V).

2.3.2 GEMS - An Industrial CEM Solver

GEMS is an abbreviation for General Electro-Magnetic Solver, which was derived as a result of a three year Swedish development project funded by an extensive research program in academia and industry. The main objective of the GEMS project was to develop a state-of-the-art software suite for solving the Maxwell equations. For time-dependent problems, GEMS uses a hybrid of Finite-Differences, and Finite-Volumes or Finite-Elements methods for solving the equations. In a typical problem an object, such as an antenna, is placed in a domain. The program then solves the equations for an incoming wave to study the radiation or scattering of the wave. In the hybrid scheme, the space close to the object is discretized using a Finite-Element or Finite-Volume method and the surrounding space is discretized using a Finite-

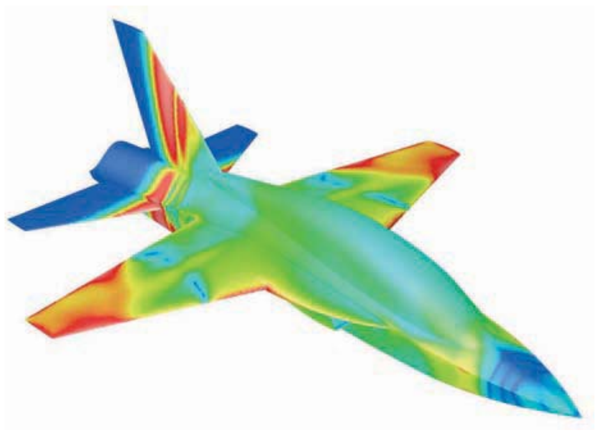


Figure 2.5: Trainer fighting jet. Surface plot of the unstructured computational grid used for the FEM discretization in GEMS

Difference method. This approach has been shown to be efficient because we can exploit the good characteristics of the different methods, see Edelvik [38] and references therein.

In papers I-III we study the Finite-Element part of GEMS. This solver consists of two parts. First, the Maxwell equations are discretized using FEM and a system of equations is assembled. Second, the linear system of equations is solved for each time step using an *iterative method* [6, 102]. Throughout this thesis we use data from a discretization of a fighter-jet geometry, see Figure 2.5. This geometry results in a very sparse linear system with 1794058 unknowns and 29870237 non-zero elements and the system is solved using our implementation of *the conjugate gradient* (CG) method [6, 116].

The performance of the CG algorithm is, in the sparse case, dominated mainly by two things: the convergence rate, how many iterations are performed, and the efficiency of implementation of the sparse matrix-vector product (SpMxV). Accelerating the convergence using *preconditioning* [6, 116] is problem-dependent out of scope of this thesis. Furthermore, for our data set, no preconditioning was necessary because the convergence rate was already sufficiently high.

Optimization of the SpMxV has been studied by several authors [17, 37, 90, 96, 101, 114, 118]. The efficiency of these optimizations depends on the distribution of the non-zero elements and the data structure used to store the iteration matrix. In the FEM case, the sparsity structure is determined by the type and numbering of the grid elements and also on the geometry of the problem. The number of non-zero elements in each row or column is bound by how many elements that geometrically couple in physical space. Figure 2.6 shows the non-zero structure of the iteration matrix used in our GEMS experiments. In the sparse matrix-vector product (SpMxV) the scalar product of each row with a dense right-hand side (RHS) is formed. It is clear from Figure 2.6 that

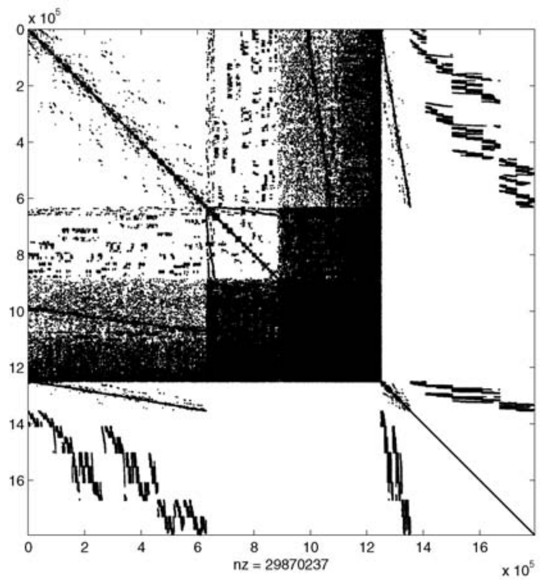


Figure 2.6: Non-zeros structure of the resulting iteration matrix for the surface grid of the Trainer fighter jet.

the access pattern of the RHS vector will exhibit a low degree of locality due to the irregular structure of the matrix. In papers I-III we study how to optimize the CG algorithm for both uniform and non-uniform shared-memory architectures. The structure of the CG algorithm is similar to other so called *Krylov subspace methods* [6, 116] and we believe that our results can be extended to this larger class of methods.

2.3.3 TGBS - A Cache Optimized Multigrid Solver

In Paper IV we study a *multigrid* solver [23, 115, 122]. Multigrid algorithms are used frequently to solve many important problems. The classic application is to solve elliptic PDE:s but multigrid methods have also been used to solve convection-diffusion problems, and for solving more complex PDE like the incompressible Navier-Stokes equations. In Paper IV we study a standard elliptic problem, the Poisson equation in 3D.

Multigrid methods are used to accelerate convergence of some iterative solvers and this acceleration is achieved by using a set of grids where the solution is interpolated between the grids and high frequency errors are suppressed using a *smoother*. Multigrid methods are characterized by an inner loop where the number of grid points used increase and decrease in a predefined way. A multigrid method involve three different components: a *prolongation* and a *restriction* operator to handle the movement of between the grids and a smoother operator to suppress the high frequency errors. The convergence rate of the method is governed by the numerical properties of these operators, see Trottenberg et al. [115], Wesseling [122]. One way of improving the convergence rate can be to apply the smoother several times on each grid level. In Paper IV we study ways of reducing the cost of applying the smoother. If the smoother is cheap to apply we can invoke it multiple times to increase convergence. The methodology we develop can be used for any structured grid multigrid problem.

2.3.4 SAMR - Structured Adaptive Mesh Refinement

In Paper V we study a structured adaptive mesh refinement (SAMR) [9, 10, 11] scheme. In SAMR methods the structured mesh is composed out of a collection of grid patches or blocks to support fine discretizations only where it is needed. The main idea is to try to resolve only the critical areas of the domain instead of using a uniform fine grid. These techniques have been used successfully to solve difficult problems, see Dongarra et al. [35]. A fundamental difference compared to a uniform grid approach is that the mesh will change dynamically as the simulation progresses. The non-static grid introduces new difficulties to the implementations because we need to maintain load balance. We can no longer use a static domain-based strategy to decompose the problem because the distribution of computational work will not be constant.

The patches or blocks are assigned to each thread to form a composite grid structure. This procedure is referred to as a *partitioning* of the grid. Load balance is achieved if the different partitions of the composite grids have approximately the same size. This load balancing strategy assumes that the computational work is proportional to the number of grid points that is assigned to each partition. For SAMR methods to be efficient the overhead associated with partitioning and applying the numerical method on composite grids need to be kept small [5, 14, 32, 33, 36, 73, 94, 98, 100, 104, 120, 124]. Furthermore, the composite grid can be seen as a union of several small PDE problems[2]. The boundary values need to be interpolated from neighboring grids. Numerically, special care has to be taken to not destroy order of accuracy and stability in the interpolation process.

On non-uniform architectures we have to take data distribution into account. In the message-passing model we must explicitly distribute patches according to the current partitioning over the processors. Several studies have been made to develop partitioning methods that try to minimize patch movement between nodes [35, 120]. On DSM systems we are faced with a similar problem. Here, all patches can be accessed but to ensure a high degree of geographical locality the physical allocation of patches must match the partitioning. To solve this problem dynamic data migration (or replication) must be used. Without migration, data will be allocated according to the page placement policy of the system. This is studied in Paper V.

[2]The actual boundary conditions of the original problem need to be applied for patches that touch the physical boundary

3. Contributions

IN THIS CHAPTER we list our contributions to the solution of the PDE problems described in Section 2.3. The chapter is divided into three sections corresponding to the three solvers studied. For each section we first comment on our implementation and then present summaries of the material presented in papers I-V.

3.1 GEMS

We parallelize the conjugate gradient algorithm, described in, Algorithm 1, using OpenMP by adding work-sharing directives to the loops that implement the compute intensive operations. We include the entire main iteration into a single parallel region to reduce the overhead of entering and leaving parallel regions. In the CG algorithm, there are several global barriers and reductions per iteration. The BLAS level-1 operations [13] are easily parallelized and load balanced using OpenMP work-share directives. However, for the SpMxV operation, it is clear from Figure 2.6 that the access pattern will exhibit very little spatial locality due to the structure of the matrix. A straight-forward OpenMP implementation of the SpMxV would add a work-share directive to a loop over the rows or columns. If a static loop scheduler is used this will lead to load balance problems because the non-zeros are non-evenly distributed. This load imbalance, can be mitigated using a dynamic loop scheduler. However, because the inner-loops are small and change in extent, dynamic scheduling might result in significant overhead.

The degree of geographical locality will be low because in the code the initialization of the matrix is performed sequentially. The matrix is the result of an *assembly process* where the unstructured grid is traversed to construct the matrix. On a DSM system with a first-touch page placement policy we could try to initialize the matrix in parallel to match the access pattern of the SpMxV. However, since most sparse matrix formats use indirect addressing we need the matrix in memory before any partitioning analysis can be performed which results in a chicken-and-egg problem.

3.1.1 Paper I

In Paper I we investigate the importance of geographical locality for a small set of scientific applications. These are the CG and MG applications from the

Algorithm 1 Method of Conjugate Gradients

```
Given an initial guess $x_0$, compute $r_0 = b - Ax_0$ and set $p_0 = r_0$.
For $k = 0, 1, \ldots$
(1)     Compute and Store $Ap_k$
(2)     Compute $< p_k, Ap_k >$
(3)     $\zeta_k = \frac{< r_k, r_k >}{< p_k, Ap_k >}$
(4)     $x_{k+1} = x_k + \zeta_k p_k$
(5)     Compute $r_{k+1} = r_k - \zeta_k Ap_k$
(6)     Compute $< r_{k+1}, r_{k+1} >$
(7)     $\eta_k = \frac{< r_{k+1}, r_{k+1} >}{< r_k, r_k >}$
(8)     Compute $p_{k+1} = r_{k+1} + \eta_k p_k$
```

NAS NPB suite [57], the GEMS solver and a realistic CFD solver [54]. We evaluate the use of parallel initialization, affinity-on-next-touch and dynamic page migration to overcome the problem of serial initialization on system using a first-touch page placement policy. In the case of the CG algorithm, an optimal parallel initialization is very hard to achieve due to the indirect addressing of the sparse matrix data structure. In this case, we show that affinity-on-next-touch on the SF15K and a the dynamic page migration engine of the Sun WildFire manages to increase the amount of geographical locality significantly. We also find that the effect of page migration differs somewhat when comparing codes from benchmarks and real applications. A possible reason for this is that the access patterns of real applications are more sparse and/or unstructured. Benchmark codes are also often evaluated using smaller data sets which reduce the effect of geographical locality because a more efficient use of cache memories can hide the remote latency.

Data distribution for the two NPB benchmarks [57] have been studied by several authors, see Bull and Johnson [16] (Sun Wildfire) and Nikolopoulos et al. [82] (SGI Origin 2000). To our knowledge Paper I is the first study of the NPB applications and the affinity-on-next-touch implementation on the Sun Fire series of systems. We also compare the performance of the NPB benchmarks to more realistic PDE solvers.

3.1.2 Paper II

In Paper II we investigate the scalability of the affinity-on-next-touch primitive. We show that the Solaris implementation is not very scalable because of overhead associated with keeping TLBs coherent. By using large pages and an affinity-on-next-touch directive, we manage to improve the performance of the CG solver with up to 160% compared to the original solver, where all data was allocated onto a single node by the first-touch page placement policy. For the GEMS application the larger page size did not effect the performance in a negative way. To our knowledge Paper II is the first study of the combination of larger page sizes and page migration.

3.1.3 Paper III

In Paper III we evaluate different algorithmic optimizations to increase the performance of the GEMS solver. Most of the optimizations are well known but they have, in most cases, only been evaluated in a message passing environment. The algorithmic optimizations studied are: *graph partitioning* [35, 59, 120], *S-Step formulation of the algorithm* [6, 24, 34] and *bandwidth minimization* [30, 44]. Apart from the algorithmic optimizations we also improved the OpenMP implementation and managed to reduce the number of global barriers from seven to three per iteration. To evaluate the efficiency of our implementation and the effect of the algorithmic optimizations, we execute the code on both a uniform (an E10K) and a non-uniform shared-memory architecture (a SF15K) using upto 28 threads.

On both the uniform and non-uniform architectures, bandwidth minimization exhibited the best performance. Apart from exhibiting good load balance the smaller bandwidth of the matrix increased the amount of data locality. Also, the number of remote accesses is reduced compared to the original algorithm. Combining bandwidth minimization with graph partitioning and the S-step formulation does not increase the performance compared to using bandwidth minimization only. The positive effect of bandwidth minimization has also been observed by Oliker et al. [90] using a SGI Origin 2000 system. However, in their study they used a smaller and more synthetic data set.

In Paper III we also compare the scalability of the CG solver on a uniform architecture to the scalability on a non-uniform architecture using the optimizations developed in Paper II. Our results show that scalability was much better on the uniform architecture compared to the non-uniform. Possible reasons for the difference in scalability are that in the non-uniform case, communication is more expensive because some of cache misses are served by remote nodes. Another possible reason can be attributed to NUMA-effects in the implementations of the synchronization primitives in the OpenMP runtime system, see Fredrickson et al. [42].

3.2 TBGS

In Paper IV we study how to improve locality for Gauss-Seidel and Successive-Over-Relaxation (SOR) [128] type smoothers for multigrid methods. The traditional way of parallelizing such smoothers is to introduce multi-color orderings [1, 102, 115, 122]. Such orderings decouple the data dependencies of the smoother to make it data parallel. However, multi-color orderings reduce data locality [105, 121]. The decoupling results in that the grid needs to be swept several times. At each sweep, only a part of the data is touched, leading to unused data that occupy cache space. Naturally ordered smoothers only sweep the grid once, resulting in a higher degree of locality.

A natural ordering has the drawback that it potentially introduces more parallel overhead than the multi-color case. When parallelizing natural ordered Gauss-Seidel smoothers we can use either a *hyperplane* or a *pipelined* approach [57, 60, 125, 127]. In 2D or 3D, hyperplane algorithms exhibit moderate speedup due to load imbalance problems. Pipelined algorithms have some scalability problems due to the need of fine-grained synchronization. To be efficient, the pipelining technique require very fast communication between the processors. Therefore, this technique is inappropriate for most contemporary message-passing and shared-memory systems.

3.2.1 Paper IV

In Paper IV we show how to combine a temporal blocking technique with a pipeline parallelization strategy. Our experimental results show that the increased degree of temporal locality allow us to apply the smoother more than once at a very low additional cost. By applying the smoother multiple times we improve the convergence of the multigrid method used. This is true even though the natural order exhibits a lower asymptotic convergence rate than the multi-color ordering. Our temporally blocked smoother executes up to 3 times faster compared to a multi-color implementation.

The pipeline parallelization technique introduces more parallel overhead than the standard multi-color technique. Our results show that on a DSM system, a SunFire 15000, the parallel efficiency drops significantly when more threads are added. A common belief is that emerging CMP/SMT architectures can tolerate more communication overhead than traditional shared-memory systems due to the possibility of communication using an on-chip cache. To study this we evaluated an integer version of the Gauss-Seidel smoother on a 32-thread based UltraSPARC-T1 system and the original floating-point version on a simulated 32-way CMP. On the UltraSPARC-T1 our results show that when more than one thread is scheduled on each core, our locality optimized algorithm exhibited chip-resource allocation problems. Adding one SMT thread did not improve performance as much as adding another core. This result was verified on the simulated 32-core system where the pipelined version scales as good as the multi-color implementation. This indeed indicates that emerging multi-core architectures are more communication-tolerant than traditional DSM designs. This fact opens up possibilities for alternative parallel algorithms. In our multigrid example, we increased locality at the cost of more communication.

3.3 SAMR

In Paper V we show that the first-touch page placement policy leads to a large amount of remote accesses for the SAMR solver. To remedy this, we introduce dynamic data migration using a affinity-on-next touch call. Because we want to minimize data movement we use algorithms designed for message-passing models to reduce the overhead of data migration.

3.3.1 Paper V

In Paper V we evaluate the efficiency of a affinity-on-next-touch directive used to increase the amount of geographical locality for the SAMR application. The SAMR algorithm uses a fixed number of grid blocks, see Lötstedt et al. [71], Steensland et al. [107]. Various algorithmic optimizations are used to minimize data movement between partitions. For example, the Jostle diffusion partitioner by Walshaw et al. [120] is used to minimize the intra-partition movement of grid blocks. We also develop a book-keeping strategy to only trigger migration for the grid blocks that are assigned to new nodes from the partitioner. For maximum thread-data affinity we align all data to page boundaries and we explicitly touch pages to bind them to the correct node using the affinity-on-next-touch directive. We believe that the same techniques can be used to increase geographical locality for more sophisticated SAMR algorithms where the amount of blocks can change dynamically [10, 11]. Our results show that we can reduce the amount of remote accesses significantly using page migration. The overhead introduced by page migration is small, and the final reduction in execution time is 40%.

4. Summary of Contributions

- We show that geographical locality has a significant effect on the performance of shared-memory implementations of iterative and adaptive PDE solvers, even for systems with small NUMA ratios.

- We show that a directive with an affinity-on-next-touch semantic is an efficient way of increasing the degree of geographical locality for both iterative and adaptive PDE solvers.

- We show that, for applications where the granularity of sharing is large, large pages can be used to reduce the overhead of page migration.

- We show that, for shared-memory implementations of iterative solvers of conjugate gradient type, bandwidth minimization of the iteration matrix can result in both a high degree of data locality and a good load balance.

- We show that we can increase the performance of multigrid smoothers for emerging shared-memory architectures by combining natural ordering, temporal cache blocking and a pipeline parallelization strategy.

- We show that CMP architectures exhibiting shared on-chip caches have the potential of being able to tolerate more communication overhead compared to contemporary many single-core multiprocessor architectures.

5. Conclusions

The big problems should have been solved when they were small

Unknown

IN THIS THESIS we evaluate and improve the performance of a set of realistic PDE solvers for shared-memory architectures. We hope that our contributions can serve as guidelines or as a set of best practices for implementations of iterative and adaptive PDE solvers. Our overall conclusions are that locality in various forms, is important, and will become more and more important to achieve maximal performance on future shared-memory systems. Based on our results we identify two major topics that need to be considered for future shared-memory systems:

- Emerging CMP architectures will feature a smaller total amount of cache memory as a larger percentage of the chip area is used to host multiple cores. In a traditional non-CMP system the total amount of cache memory grows as we increase the number of threads because each CPU has a private hierarchy of caches. On a typical CMP, parts of the memory hierarchy will be shared which result in that the total amount of cache memory is *constant*. We also anticipate that the CPU-memory gap will increase as the technology advances further. Both of these developments will fuel the need for locality optimizations further.
- The bandwidth demand of processors is increasing steadily. This is especially true for CMP architectures where multiple threads have to be fed by the memory system. To build scalable architectures the processors will probably have to be distributed over nodes forming non-uniform architectures. Hence, we need mechanisms to control data distribution for maximal performance.

A consequence of these developments is that the importance of locality cannot be underestimated. As an example, the methodology we develop in Paper IV show that perfect scalability may not be the only target for algorithm development for emerging shared-memory systems. Data locality has a much more prominent effect on performance. We also believe that the need for data locality might require re-evaluation of many important algorithms such as matrix factorizations and fast transforms. However, the optimization of such algorithms for future shared-memory architectures is still an open research issue.

5.1 Outlook

We have reached a point where we need to incorporate modern software development techniques to cope with the increased size and complexity of the implementations of scientific software. By hiding complexity, scientists from application fields can be more productive. This is especially true for more challenging applications. Furthermore, with the introduction of chip-multiprocessing we believe that parallel programming will be absolute necessary for achieving high performance not only for applications in the traditional HPC domain. Both of these two developments suggest that the shared-memory model will become more widespread simply because it is easy to use. Hence, we need efficient, easy to use and portable tools for handling data distribution and for optimizing locality.

Distributed shared-memory architectures were designed to meet three design goals: scalability, ease of use and low cost. However, most of the large scale systems used today can not be programmed in the shared-memory model. Developers are still using message passing to reach the highest possible performance. One fundamental problem with the DSM model is the problem of artifactual communication and data distribution. Whether these problems can be solved still remains an open research issue. One interesting idea is to implement the coherence protocols in software. These so-called software distributed shared-memory systems (SW-DSM) [70, 97] can be used to implement the shared-memory model on a clustered architecture. However, SW-DSMs have not yet reached the same maturity and level of performance as their hardware counterparts.

In papers I,II and V we study a directive with an affinity-on-next-touch semantic. Such a directive is architecture-agnostic which leaves a programming model such as OpenMP virtually untouched[1]. We show that this type of directive can solve many data distribution problems and we believe that for large-scale applications we some mechanism to control data distribution. In this setting an affinity-on-touch directive looks quite promising. In our experiments we disabled the OS scheduler by binding threads to CPU:s. In a realistic production environment where the system can be over-prescribed we need affinity aware scheduling in combination with data distribution. An affinity-on-touch directive could serve as a hint to improve thread-data affinity.

Some of the data distribution experiments in the papers are performed at a small scale to be able to compare with a corresponding UMA system. When more threads and nodes are used we find that the overhead for page migra-

[1]For uniform architectures such a directive would be rendered as NULL which may not be the most esthetic solution

tion becomes more of an issue. This overhead has also been identified as a significant bottleneck by other researchers [56, 65, 66, 117]. We need to find new ways of scaling page migration to support larger systems. As an example, consider the case of shared TLBs for SMT processors. If one thread needs to shoot-down a TLB entry, other threads may have to wait [29, 72, 99]. Various algorithmic optimizations could also be developed to minimize the amount of migration. For some applications, where the granularity of sharing allows it, larger pages could help reducing this type of overhead.

6. Acknowledgements

FIRST of all I wish to thank myself for taking the opportunity to start my PhD studies and also for completing the thesis. The years in Uppsala have been very rewarding for me and I will remember this period for as long as I live.

Second, I wish to thank my advisors, Jarmo, Sverker and Erik. Without your excellent support and expertise I would not have completed this thesis. I also want to thank my co-authors Markus, Dan and Zoran for superb collaborational skills both back home at Pollax and when attending conferences elsewhere in the world. Special thanks also goes to the glorious UART team: Cap'n Erik, Lalle, Martin, Zoran, Dan, Erik and Håkan. *Yarr! Weigh anchor! Hoist the mizzen! Savvy, ye scallywaggys!*

Third, I want to acknowledge the support from my family and friends. Although you haven't directly contributed to the thesis you have given me the necessary distractions (love, warmth, food, good music and money) needed for very deep thoughts. Thank you.

Fourth, I want to thank all the people at the department contributing to a very happy and productive atmosphere. Special thanks goes to the forgotten heros down in the sys-admin dungeons: Jukka, Janne and Joel.

Finally my most sincere thanks goes to my darling Camilla for being the most adorable human being on the planet. Puss på dig!

This work was in part supported by the Parallel and Scientific Computing Institute (PSCI) of Sweden and Sun Microsystems.

Bibliography

[1] L. M. Adams and J. M. Ortega. A Multi-Color SOR Method for Parallel Computation. In *Proceedings of the International Conference on Parallel Processing*, pages 53–58, 1982.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259. ACM Press, 2000. ISBN 1-58113-232-8. URL `http://doi.acm.org/10.1145/339647.339691`.

[3] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Co mputer capabilities. In *AFIPS Conference Proceedings*, 1967.

[4] E. Artiaga, N. Navarro, X. Martorell, and Y. Becerra. Implementing PARMACS Macros for Shared-Memory Multiprocessor Environments. Technical Report UPC-DAC-1997-07, Department of Computer Architecture, Polytechnic University of Catalunya, Jan. 1997.

[5] D. Balsara and C. Norton. Highly parallel structured adaptive mesh refinement using parallel language-based approaches. *Parallel Computing*, 27:37–70, 2001.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[7] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 282–293. ACM Press, 2000. ISBN 1-58113-232-8. URL `http://doi.acm.org/10.1145/339647.339696`.

[8] B. Beck. Shared-memory parallel programming in c++. *IEEE Softw.*, 7(4):38–48, 1990. ISSN 0740-7459. URL `http://dx.doi.org/10.1109/52.56449`.

[9] M. Berger. Data structures for adaptive grid generation. *SIAM Journal of Scient. Stat. Comput*, 7:904–916, Jul 1986.

[10] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, May 1989.

[11] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, Mar 1984.

[12] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA machines. *Scientific Programming*, 8:163–181, 2000.

[13] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002. ISSN 0098-3500. URL `http://doi.acm.org/10.1145/567806.567807`.

[14] R. Blikberg. *Nested Parallelism in OpenMP with Application to Adaptive Mesh Refinement.* PhD thesis, Parallab/Department of Informatics, University of Bergen, Norway, Februari 2003.

[15] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-701-6. URL `http://doi.acm.org/10.1145/209936.209958`.

[16] J. M. Bull and C. Johnson. Data Distribution, Migration and Replication on a cc-NUMA Architecture. Technical report, UKHEC, 2002. Also in: Proceedings of the Fourth European Workshop on OpenMP (EWOMP) 2002.

[17] D. A. Burgess and M. B. Giles. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Adv. Eng. Softw.*, 28(3):189–201, 1997. ISSN 0965-9978. URL `http://dx.doi.org/10.1016/S0965-9978(96)00039-7`.

[18] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, George Washington University, May 13 1999. Second Printing.

[19] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 12–24. ACM Press, 1994. ISBN 0-89791-660-3. URL `http://doi.acm.org/10.1145/195473.195485`.

[20] R. Chandra, D.-K. Chen, R. Cox, D. E. Maydan, N. Nedeljkovic, and J. M. Anderson. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 334–345. ACM Press, 1997. ISBN 0-89791-907-6. URL `http://doi.acm.org/10.1145/258915.258945`.

[21] A. Charlesworth. The sun fireplane system interconnect. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 7–7. ACM Press, 2001. ISBN 1-58113-293-X. URL `http://doi.acm.org/10.1145/582034.582041`.

[22] A. Chauhan, B. Sheraw, and C. Ding. Scalability and data placement on SGI Origin. Technical Report TR98-305, Rice University, Houston, TX, Apr. 1998.

[23] E. Chow, R. Falgout, J. Hu, R. Tuminaro, and U. Yang. A Survey of Parallelization Techniques for Multigrid Solvers. Technical Report UCRL-BOOK-205864, LLNL, 2004.

[24] A. Chronopoulos and C. Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[25] I. P. A. S. Committee. Portable operating system interface (posix)–part1: System application programming interface (api) [c language]. *IEEE Std 1003.1-1996, ISO/IEC 9945-1*, 1996.

[26] J. Corbalan, X. Martorell, and J. Labarta. Evaluation of the memory page migration influence in the system performance: the case of the sgi o2000. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 121–129. ACM Press, 2003. ISBN 1-58113-733-8. URL `http://doi.acm.org/10.1145/782814.782833`.

[27] J. Corbalan, X. Martorell, and J. Labarta. Page migration with dynamic space-sharing scheduling policies: the case of the sgi 02000. *Int. J. Parallel Program.*, 32(4):263–288, 2004. ISSN 0885-7458. URL `http://dx.doi.org/10.1023/B:IJPP.0000035815.13969.ec`.

[28] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1998.

[29] M. Curtis-Maury, T. Wang, C. Antonopoulos, and D. Nikolopoulos. Integrating multiple forms of multithreaded execution on multi-smt systems: A study with scientific applications. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 199–209, Los Alamitos, CA, USA, 2005. IEEE Computer Society. ISBN 0-7695-2427-3. URL `http://doi.ieeecomputersociety.org/10.1109/QEST.2005.16`.

[30] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM Press, 1969.

[31] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science and Engineering, IEEE*, 5(1):46–55, Jan.-March 1998.

[32] R. Deiterding. Construction and application of an amr algorithm for distributed memory computers. In *Adaptive Mesh Refinement – Theory and Applications, Proc. of the Chicago Workshop on Adaptive Mesh Refinement Methods*, pages 361–372. Springer, 2003.

[33] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challanges in dynamic load balancing. *Appl. Numer. Math.*, 52(2-3):133–152, 2005. ISSN 0168-9274. URL `http://dx.doi.org/10.1016/j.apnum.2004.08.028`.

[34] J. Dongarra and V. Eijkhout. Finite-choice algorithm optimization in conjugate gradients. Computer Science Report UT-CS-03-502, Universtity of Tennesse, 2003. LAPACK Working Note 159.

[35] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2003.

[36] J. Dreher and R. Grauer. Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws. *Parallel Computing*, 31:913–932, 2005.

[37] I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002. ISSN 0098-3500. URL `http://doi.acm.org/10.1145/567806.567810`.

[38] F. Edelvik. *Hybrid solvers for the Maxwell equations in time-domain*. Uppsala dissertations form the Faculty of Science and Technology. Acta Universitatis Upsaliensis, 2002. ISBN 91-554-5354-6. URL `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-2156`.

[39] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997. ISSN 0272-1732. URL `http://dx.doi.org/10.1109/40.621209`.

[40] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003. ISBN 0471220485.

[41] B. Falsafi and D. A. Wood. Reactive numa: a design for unifying s-coma and cc-numa. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 229–240, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-901-7. URL `http://doi.acm.org/10.1145/264107.264205`.

[42] N. R. Fredrickson, A. Afsahi, and Y. Qian. Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 140–149. ACM Press, 2003. ISBN 1-58113-733-8. URL `http://doi.acm.org/10.1145/782814.782835`.

[43] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and design of alphaserver gs320. *ACM SIGPLAN Notices*, 35(11):13–24, 2000. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/356989.356991`.

[44] N. E. Gibbs, J. William G. Poole, and P. K. Stockmeyer. An Algorithm for Reducing the Bandwith and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, April 1976.

[45] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, Los Alamitos, CA, USA, 1983. IEEE Computer Society Press. ISBN 0-89791-101-6.

[46] A. Grama, A. Gupta, G. Karypsis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.

[47] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, Feb. 1999.

[48] E. Hagersten, A. Landin, and S. Haridi. Ddm: A cache-only memory architecture. *Computer*, 25(9):44–54, 1992. ISSN 0018-9162. URL `http://dx.doi.org/10.1109/2.156381`.

[49] E. Hagersten, A. Saulsbury, and A. Landin. Simple coma node implementations. In *System Sciences, 1994. Vol. I: Architecture, Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 1, pages 522–533, January 1994.

[50] J. L. Hennessy and D. A. Patterson. *Computer Architecture;A Quantative Approach*. Morgan Kaufman, 3rd edition, 2003.

[51] S. Holmgren, M. Nordén, J. Rantakokko, and D. Wallin. Performance of PDE Solvers on a Self-Optimizing NUMA Architecture. *Parallel Algorithms and Applications*, 17(4):285–299, 2002.

[52] C. Holt, J. P. Singh, and J. Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 134–145, New York, NY, USA, 1996. ACM Press. ISBN 0-89791-786-3. URL `http://doi.acm.org/10.1145/232973.232988`.

[53] R. Iyer, H. Wang, and L. N. Bhuyan. Design and analysis of static memory management policies for cc-numa multiprocessors. *J. Syst. Archit.*, 48(1-3):59–80, 2002. ISSN 1383-7621. URL `http://dx.doi.org/10.1016/S1383-7621(02)00066-8`.

[54] A. Jameson and D. A. Caughey. How many steps are required to solve the euler equations of steady, compressible flow - in search of a fast solution algorithm. In *AIAA Computational Fluid Dynamics Conference, 15th, Anaheim, CA, June 11-14*, 2001. AIAA-2001-2673.

[55] D. Jiang and J. P. Singh. A methodology and an evaluation of the sgi origin2000. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 171–181, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-982-3. URL `http://doi.acm.org/10.1145/277851.277902`.

[56] D. Jiang and J. P. Singh. Scaling application performance on a cache-coherent multiprocessor. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 305–316, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0170-2. URL `http://doi.acm.org/10.1145/300979.301005`.

[57] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[58] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[59] G. Karypsis and V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[60] T. Kim and C.-O. Lee. A Parallel Gauss-Seidel Method using NR Data Flow Ordering. *Applied Mathematics and Computation*, 99(2):209–220, 1999.

[61] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

[62] K. Krewell. Power5 Tops on Bandwidth. *Microprocessor Report*, December 22, 2003.

[63] K. Krewell. Sparc Turns 90 nm. *Microprocessor Report*, October 25, 2004.

[64] K. Krewell. Double Your Opterons; Double Your Fun. *Microprocessor Report*, October 11, 2004.

[65] A.-C. Lai and B. Falsafi. Comparing the effectiveness of fine-grain memory caching against page migration/replication in reducing traffic in dsm clusters. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 79–88, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-185-2. URL `http://doi.acm.org/10.1145/341800.341811`.

[66] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 241–251. ACM Press, 1997. ISBN 0-89791-901-7. URL `http://doi.acm.org/10.1145/264107.264206`.

[67] H. Löf and S. Holmgren. affinity-on-next-touch: increasing the performance of an industrial pde solver on a cc-numa system. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 387–392, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-167-8. URL `http://doi.acm.org/10.1145/1088149.1088201`.

[68] H. Löf and J. Rantakokko. Algorithmic Optimizations of a Conjugate Gradient Solver on Shared Memory Architectures. *Internation Journal of Parallel, Emergent and Distributed Systems*, 21(5):345–363, October 2006. Journal formerly known as Parallel Algorithms and Applications.

[69] H. Löf, M. Nordén, and S. Holmgren. Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers. In M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra,

editors, *Computational Science - ICCS 2004; 4:th International Conference, Krakòw, Poland, June 6-9, 2004, Proceedings, Part II*, volume 3037/2004, pages 9–16. Springer Berlin/Heidelberg, 2004. ISBN 3-540-22115-8.

[70] H. Löf, Z. Radović, and E. Hagersten. THROOM - Supporting POSIX Multithreaded Binaries on a Cluster. In *Euro-Par 2003 Parallel Processing*, volume Volume 2790/2004 of *Lecture Notes in Computer Science*, pages 760–769, Klagenfurt, Austria, May 2004. Springer Berlin / Heidelberg. ISBN 3-540-40788-X. doi: 10.1007/b12024.

[71] P. Lötstedt, S. Söderberg, A. Ramage, and L. Hemmingsson-Frändén. Implicit solution of hyperbolic equations with space-time adaptivity. *BIT*, 42(1):134–158, 2002.

[72] P. Mackerras, T. S. Mathews, and R. C. Swanberg. Operating system exploitation of the power5 system. *IBM Journal of Research and Development*, 49(4/5):533–539, July/September 2005.

[73] P. MacNeice. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer physics communications*, 126:330–354, 2000.

[74] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–99, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-189-9. URL `http://doi.acm.org/10.1145/1122971.1122987`.

[75] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. *ipps*, 00:480, 1995. ISSN 1063-7133. URL `http://doi.ieeecomputersociety.org/10.1109/IPPS.1995.395974`.

[76] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, 6(1), Feb 2002.

[77] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood. Improving multiple-cmp systems using token coherence. In *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)*, pages 328–339, Los Alamitos, CA, USA, 2005. IEEE Computer Society. URL `http://doi.ieeecomputersociety.org/10.1109/HPCA.2005.17`.

[78] T. G. Mattson. How good is OpenMP. *Scientific Programming*, 11: 81–93, 2003.

[79] J. McGregor. Ringside for 2006 Dual-Core Fights. *Microprocessor Report*, December 19, 2005.

[80] D. S. Nikolopoulos and E. Ayguadé. A study of implicit data distribution methods for openmp using the spec benchmarks. In *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*, volume 2104 of *Lecture Notes in Computer Science*, pages 115–129, London, UK, 2001. Springer-Verlag. ISBN 3-540-42346-X.

[81] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A case for user-level dynamic page migration. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 119–130, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-270-0. URL `http://doi.acm.org/10.1145/335231.335243`.

[82] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A transparent runtime data distribution engine for OpenMP. *Scientific Programming*, 8:143–162, 2000.

[83] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is data distribution necessary in openmp? In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 47, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

[84] D. S. Nikolopoulos, E. Ayguadé;, T. S. Papatheodorou, C. D. Polychronopoulos, and J. Labarta. The trade-off between implicit and explicit data distribution in shared-memory programming paradigms. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 23–37, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-410-X. URL `http://doi.acm.org/10.1145/377792.377801`.

[85] D. S. Nikolopoulos, E. Ayguadé;, T. S. Papatheodorou, C. D. Polychronopoulos, and J. Labarta. Scheduler-activated dynamic page migration for multiprogrammed dsm multiprocessors. *Journal of Parallel and Distributed Computing*, 62(6), June 2002.

[86] D. S. Nikolopoulos, E. Ayguadé;, and C. D. Polychronopoulos. Runtime vs. manual data distribution for architecture-agnostic shared-memory programming models. *Int. J. Parallel Program.*, 30(4):225–255, 2002. ISSN 0885-7458. URL `http://dx.doi.org/10.1023/A:1019899812171`.

[87] L. Noordergraaf and R. van der Pas. Performance experiences on Sun's Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 38. ACM Press, 1999. ISBN 1-58113-091-0. URL http://doi.acm.org/10.1145/331532.331570.

[88] M. Nordén, H. Löf, J. Rantakokko, and S. Holmgren. Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. Technical Report 2006-038, Department of Information Technology, Uppsala Universitet, Uppsala, Sweden, August 2006.

[89] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998. ISSN 1061-7264. URL http://doi.acm.org/10.1145/289918.289920.

[90] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review*, 44(3):373–393, 2002.

[91] K. Olukotun and L. Hammond. The Future of Microprocessors. *ACM Queue*, 3(7):26–29, September 2005. ISSN 1542-7730. URL http://doi.acm.org/10.1145/1095408.1095418.

[92] *OpenMP Fortran Specification v2.5*. OpenMP Architecture Review Board, May 2005. URL http://www.openmp.org.

[93] R. A. Overbeek and J. Boyle. *Portable Programs for Parallel Processors*. Saunders College Publishing, Philadelphia, PA, USA, 1987. ISBN 0030144043.

[94] M. Parashar and J. Browne. System engineering for high performance computing software: The hdda/dagh infrastructure for implementation of parallel structured adaptive mesh refinement. In *IMA Volume on Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, pages 1–18, 2000.

[95] E. Pärt-Enander. *Overlapping Grids and Applications in Gas Dynamics*. PhD thesis, Uppsala University, Department of Scientific Computing, May 1995.

[96] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 30. ACM Press, 1999. ISBN 1-58113-091-0. URL http://doi.acm.org/10.1145/331532.331562.

[97] Z. Radovic. *Software Techniques for Distributed Shared Memory*. PhD thesis, Uppsala Universitet, 2005.

[98] J. Rantakokko. Partitioning strategies for structured multiblock grids. *Parallel Computing*, 26:1661–1680, 2000.

[99] J. A. Redstone, S. J. Eggers, and H. M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 245–256, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-317-0. URL `http://doi.acm.org/10.1145/378993.379245`.

[100] C. A. Rendleman. Parallelization of structured, hiearchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3:147–157, 2000.

[101] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[102] Y. Saad and H. A. van der Vorst. Iterative Solution of Linear Systems in the 20th Century. *Journal of Computational and Applied Mathematics*, 123(1-2):1–33, Now 2000.

[103] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple coma. In *HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 276, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6445-2.

[104] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. *sc*, 00:59, 2000. ISSN 1063-9535. URL `http://doi.ieeecomputersociety.org/10.1109/SC.2000.10035`.

[105] S. Sellappa and S. Chatterjee. Cache-Efficient Multigrid Algorithms. *International Journal of High Performance Computing Applications*, 18 (1):115–133, 2004.

[106] V. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. Flexible use of memory for replication/migration in cache-coherent dsm multiprocessors. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 342–355, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8491-7. URL `http://doi.acm.org/10.1145/279358.279403`.

[107] J. Steensland, S. Söderberg, and M. Thuné. A comparison of partitioning schemes for blockwise parallel samr algorithms. In *PARA '00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, volume

1947 of *Lecture Notes in Computer Science*, pages 160–169, London, UK, 2001. Springer-Verlag. ISBN 3-540-41729-X.

[108] *Solaris Memory Placement Optimization and Sun Fire servers.* Sun Microsystems, January 2003.

[109] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2006.

[110] H. Sutter and J. Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, September 2005. ISSN 1542-7730. URL `http://doi.acm.org/10.1145/1095408.1095421`.

[111] P. J. Teller. Tranlation-Lookaside Buffer Consistency. *Computer*, 23(6):26–36, 1990. ISSN 0018-9162. URL `http://dx.doi.org/10.1109/2.55498`.

[112] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan. 2002.

[113] M. M. Tikir and J. K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3. URL `http://dx.doi.org/10.1109/SC.2004.64`.

[114] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. Res. Develop*, 41(6):711–725, 1997.

[115] U. Trottenberg, C. Oosterlee, and A. Schuller. *Multigrid*. Academic Press, 2001.

[116] H. A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems*. Cambridge monographs on applied and computational mathematics, 2003.

[117] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 279–289. ACM Press, 1996. ISBN 0-89791-767-7. URL `http://doi.acm.org/10.1145/237090.237205`.

[118] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–35. IEEE Computer Society Press, 2002.

[119] D. Wallin, H. Löf, E. Hagersten, and S. Holmgren. Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors. In *Proceedings of the 2006 International Conference on Supercomputing*, pages 145–155, New York, NY, USA, June 2006. ACM Press. ISBN 1-59593-282-8.

[120] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Parallel Distributed Computing*, 47(2):102–108, 1997.

[121] C. Weiss, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the Conference on Supercomputing*, page 31, 1999. ISBN 1-58113-091-0.

[122] P. Wesseling. *An Introduction to Multigrid Methods.* John Wiley and Sons Ltd., Chichester, 1992.

[123] K. M. Wilson and B. B. Aglietti. Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 33–33, New York, NY, USA, 2001. ACM Press.

[124] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott. Large scale parallel structured amr calculations using the samrai framework. *sc*, 00:22, 2001. URL `http://doi.ieeecomputersociety.org/10.1109/SC.2001.10029`.

[125] M. Yarrow and R. V. der Wijngaart. Communication Improvement for the LU NAS Parallel Benchmark: A Model for Efficient Parallel Relaxation Schemes. NAS Technical Report NAS-97-032, NASA Ames Research Center, 1997.

[126] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance java dialect. *Concurrency: Practice and Experience*, September-November 1998.

[127] S. Yoon, G. Jost, and S. Chang. Parallelization of Gauss-Seidel Relaxation for Real Gas Flow. NAS Technical Report NAS-05-011, NASA Ames Research Center, 2005.

[128] D. M. Young. *Iterative Solution of Large Linear Systems.* Academic Press, New York, 1971.

Distribution: publications.uu.se
urn:nbn:se:uu:diva-7136

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2006